



XAPP482 (v1.0) August 19, 2004

MicroBlaze Platform Flash/PROM Boot Loader and User Data Storage

Author: Shalin Sheth

Summary

This application note describes a working MicroBlaze™ system that stores software code, user data, and configuration data in non-volatile Platform Flash PROMs, simplifying system design and reducing cost. This application note further provides a portable hardware design, software design, and additional script utilities to be used during the implementation flow.

Introduction

Many FPGA designs that incorporate software embedded systems using MicroBlaze and PowerPC™ processors also utilize external volatile memory to execute software code. Systems using volatile memory must also include a non-volatile device to store the software code during power-down. Most FPGA systems include a Platform Flash PROM, herein referred to as PROM, on the board to load the FPGA configuration data upon power-up. Additionally, many applications might use other non-volatile devices (for example, SPI Flash, Parallel Flash, or PICs) to hold small amounts of user data, such as MAC addresses, leading to a large number of non-volatile devices on a system board.

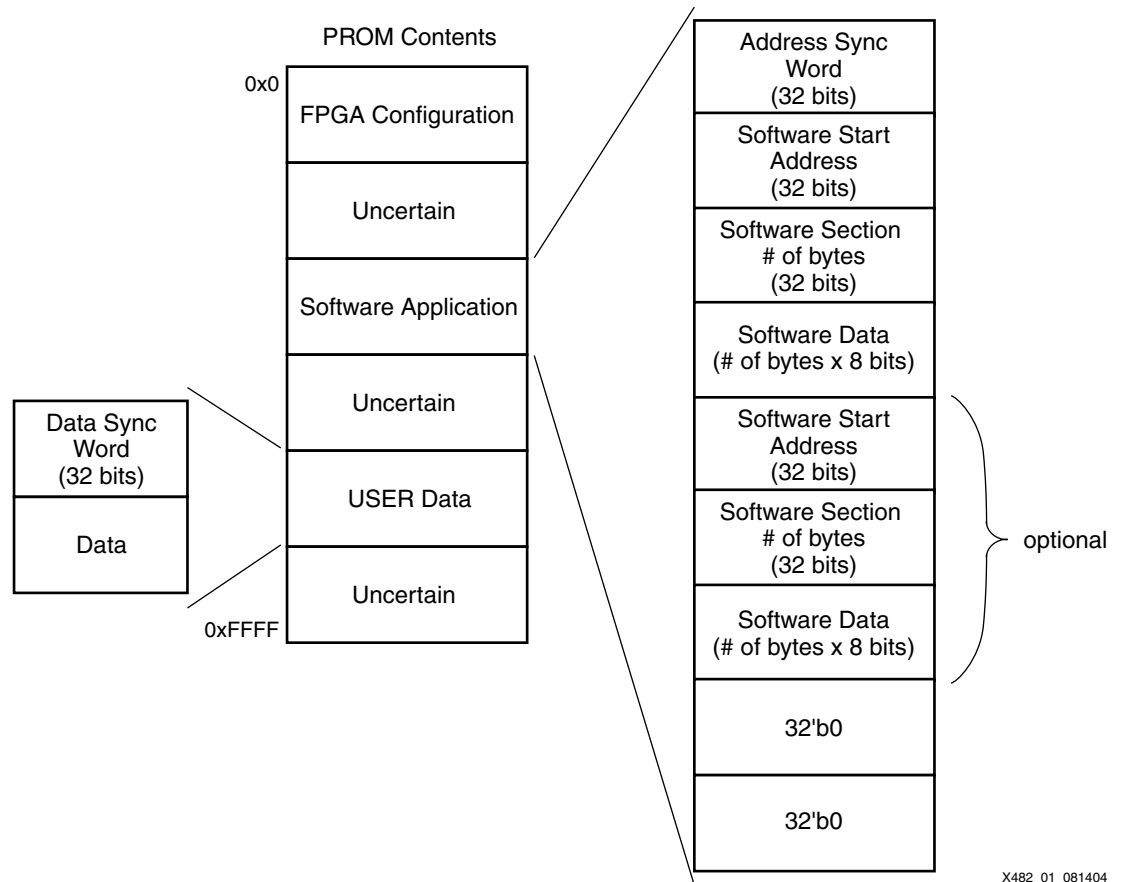
This application note demonstrates how to reduce the need for multiple non-volatile devices on the system board and how to use one PROM for FPGA configuration data, software code, and user data. The concepts covered in this application note are:

- Loading applications software from PROMs
- Introducing methods of storing multiple blocks of data in PROMs as shown in [Figure 1](#)
- Building minimal MicroBlaze memory systems for applications such as boot loading
- Introducing dynamic rewriting of reset, interrupt, and exception vectors in C code
- Defining software flows for appending a PROM file with software and user data

Although this application note is written for the low-cost MicroBlaze embedded processor core, it is portable to any 8-bit, 16-bit, or 32-bit microcontroller with general-purpose input/output ports.

© 2004 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at <http://www.xilinx.com/legal.htm>. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.



X482_01_081404

Figure 1: Methods of Storing Multiple Data Sections in a PROM

Figure 1 shows the contents of a PROM when multiple sections are stored. The software application section can be anywhere in the PROM and is identified by the address sync word. Following the address sync is a 32-bit software start address, a 32-bit software section specifying the number of bytes to follow, and then the actual software data. The software start address, number of bytes, and additional software data can be repeated multiple times in the same software application. The end of the software application section is defined by two 32-bit words equaling zero. The USER data section is defined simply by a USER sync word followed by data. The data between any FPGA configuration data, software application, or USER data is uncertain, which is the reason that the synchronization words are used.

Board Considerations

To read the PROM after the FPGA has been configured, some requirements must be considered when designing the system board. This section describes the Master Serial configuration connections and the reasons for the necessary connections.

Figure 2 shows the board connections that are necessary in a Master Serial configuration method. For more details, refer to [XAPP694](#), whose board considerations are the same as for this reference design.

The Xilinx Spartan™-3 Starter Kit board provides a working example with these board considerations. A description of this board and its schematics are available in [UG130](#).

For more information regarding FPGA configuration, refer to [XAPP501](#) and [XAPP138](#).

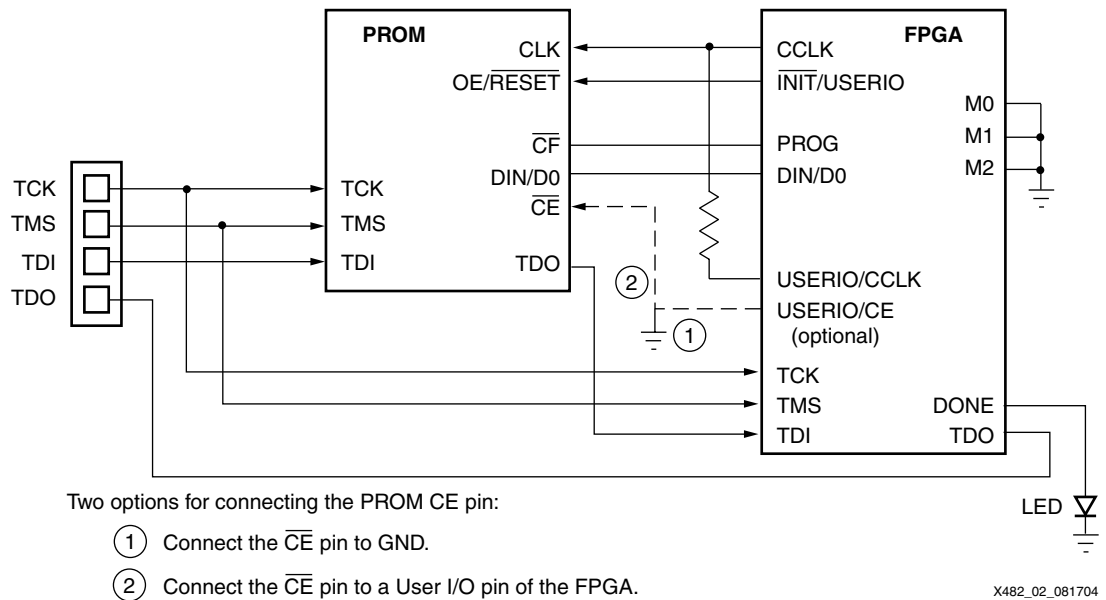


Figure 2: Board Considerations

PROM \overline{CE} Pin

The PROM \overline{CE} pin is usually connected to the DONE pin of the FPGA to hold the PROM in standby after the FPGA is configured. This pin allows the user to enable or disable the PROM and reduce power consumption when the PROM is not going to be accessed. However, if the DONE pin is connected to the PROM \overline{CE} pin, then the PROM cannot be read after FPGA configuration.

There are two options on how to connect the PROM \overline{CE} pin:

1. Connect the \overline{CE} pin to GND.
2. Connect the \overline{CE} pin to a User I/O pin of the FPGA. This option requires an additional I/O pin to the solution; however, it allows the PROM to be put into standby mode to allow power saving. The software drives this pin to enable or disable it. The Platform Flash maximum standby current is 1 mA, and the maximum active current is 10 mA (see [DS123](#) for more details).

When the FPGA DONE pin is disconnected from the PROM \overline{CE} pin, the FPGA DONE pin can be connected to an external LED to show when the FPGA has been configured (see [Figure 2](#)).

PROM CLK Pin

Route an additional User I/O pin from the FPGA to drive the CLK input of the PROM. This connection is required for this reference design because in any master configuration mode the configuration clock CLK generated by the FPGA stops toggling after the FPGA is successfully configured, preventing the PROM's address counter from advancing beyond the FPGA design stored in the PROM. The additional User I/O clocks the CLK pin of the PROM when the PROM is read after FPGA configuration. There is a 390 Ω resistor on this trace to avoid contention between two possible drivers of the CLK signal.

PROM OE / \overline{RESET} Pin

Connect the PROM OE/ \overline{RESET} pin to the FPGA \overline{INIT} pin to allow the FPGA to re-initiate a configuration if a CRC error occurs during configuration. The \overline{INIT} pin becomes a User I/O after configuration, and thus can be configured to output a logic High to keep the PROM outputs enabled.

PROM DIN/D0 Pin

Connect the FPGA DIN/D0 pin to the PROM DIN/D0 pin for the PROM data to be read into the FPGA. This is not a special connection for this application, and the DIN pin is not available for User I/O after configuration.

Hardware Design

To implement this reference design, a MicroBlaze system is used in the Embedded Development Kit (EDK). The hardware core is built on a simple On-chip Peripheral Bus (OPB) general-purpose input/output (GPIO) core to control the \overline{INIT} , \overline{CE} , OE, and DIN pins, which are described in “Board Considerations.” Figure 3 shows a block diagram of the hardware system.

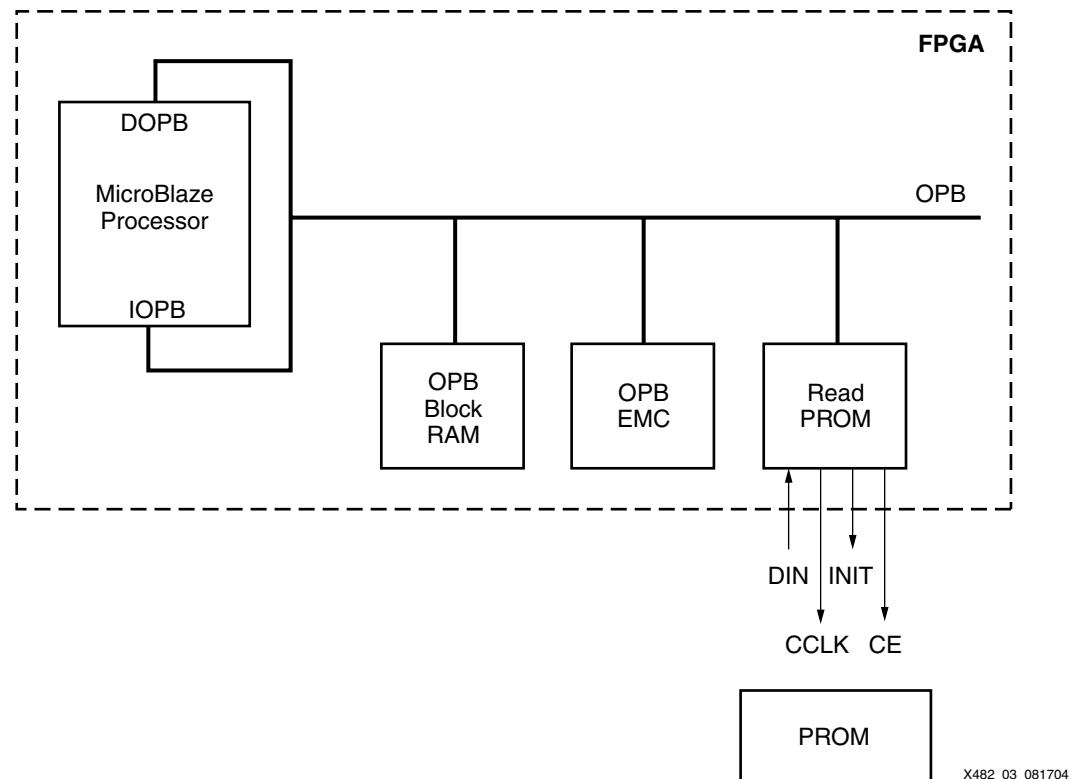


Figure 3: MicroBlaze Hardware System Block Diagram

The promread GPIO core uses 26 four-input Look Up Tables (LUTs) and 61 flip-flops in a Spartan-3 device. Additionally, this reference design uses a custom OPB Block RAM interface controller core where only one block RAM is used to show a minimal system. The minimal system in EDK 6.2i always uses four block RAMs. In an optimized system where that much block memory is not needed, such as a boot loader, a custom block RAM interface controller core is used to create a system with a single block RAM.

A possible enhancement to this reference design is to use the Master Parallel configuration implementation. A Master Parallel implementation requires the addition of seven dedicated I/Os, and it decreases the time needed to access the PROM by a factor of eight.

Firmware Design

The complexity of this application is in the firmware design. The control of the PROM is handled through the C software program. Configuration PROMs are non-addressable storage elements, where all data is sent out serially and read by the software system.

Driver Basics

The promread function handles the interaction with the PROM:

```
Xuint32 promread (Xuint8 read)
```

Table 1 lists the two modes that the PROM can be read. In an address section read, the function copies software code from the PROM to the memory location described after the address synchronization word (0x9F8FAFBF by default) in the PROM. In a data word read, the first 32-bit data word is read after the data synchronization word (0x8F9FAFBF by default).

Table 1: promread Function Description

Function	Input (read)	Output (returns)
Address section read	0x1	Always 0x0
Data word read	0x0	32-bit data word following data sync word

Figure 4 shows examples of the usage of these two functions.

```
#define DATAREAD 0x0
#define ADDRREAD 0x1

//to copy the contents from the PROM to a memory location
promread(ADDRREAD);

//to return a 32-bit word stored after the sync word.
xil_printf("\n\rData %x\n\r", promread(DATAREAD));
```

Figure 4: Example Usage of the promread Function

The Perl script, which is used to populate the MCS file (Xilinx's extension to the Intel extended hexadecimal format), checks for the uniqueness of the synchronization word. Refer to ["Usage/Flow"](#) for more information on the Perl script. The ["Driver Details"](#) section discusses how the PROM data is constructed.

Driver Details

To understand the firmware design, it is critical to understand the contents of the PROM that are serially output from the PROM. [Figure 1, page 2](#) shows how the data can be stored in the PROM.

Address Read

When an address read is indicated (promread input = 0x1), then the software reads through the PROM 32 bits at a time until it finds a 32-bit word matching the address synchronization word. For information on how to read data from the PROM, see ["Reading the PROM."](#) The default address synchronization word is 0x9F8FAFBF; however, it can be changed in the `promread.h` header file and the Perl script that populates the MCS file. The uniqueness of the synchronization word can be confirmed when the PROM file is created (refer to ["Usage/Flow"](#) for more information).

Figure 5 shows how the data is found in the PROM. Once an address synchronization word is found, immediately following it is a 32-bit word representing the starting address of where the software code is stored in the processor memory map. Following this address is the number of bytes to store after the start address. The software then reads the number of bytes from the PROM and copies data to the address specified at the start of the block. At the completion of the first address section, the software reads two 32-bit words. If either value is greater than zero, then the first 32-bit word is the starting address of the next software data section and the

second word is the number of bytes in the next address section. The software continues to read address sections until reaching the end-of-address sequence of two 32-bit words equaling zero as shown in [Figure 1, page 2](#). The promread function continues to read the PROM and copy address sections as described in this paragraph until the END_PROM word of 0xFFFFFFFF is read from the PROM. Then the promread function returns a 0x0. The value of 0xFFFFFFFF indicates that the start of the blank data in the PROM is reached.

9F8FAFBF	80180000	00007100	BA101056
-----	-----	-----	-----
address	memory	number	data
sync	mapped	of	...
word	starting	bytes	
	address		

Figure 5: Example Contents of PROM for Software Section

Data Read

When a data read is indicated (the promread input = 0x0), then the software reads through the PROM 32 bits at a time until it finds a 32-bit word matching the data synchronization word. The default data synchronization word is 0x8F9FAFBF; however, it can be changed in the `promread.h` header file and the Perl script that populates the MCS file. The uniqueness of the synchronization word can be confirmed when the PROM file is created (refer to [“Usage/Flow”](#) for more information). Once the data synchronization word is found, then the promread function returns the first 32-bit word following the data synchronization word. If additional data is needed, you must modify the data retrieval section of the promread function to your requirements.

Reading the PROM

To read the PROM in software, the MicroBlaze GPIO toggles the CLK pin of the PROM. The INIT pin must be High and the CE pin must be Low to enable the PROM for reading. Every byte that comes out of the PROM is bit swapped.

```
//clock the PROM to output data
XIo_Out32(XPAR_PROMREAD_BASEADDR, OE_HIGH | CCLK_HIGH | CE_LOW);
XIo_Out32(XPAR_PROMREAD_BASEADDR, OE_HIGH | CCLK_LOW | CE_LOW);
```

Figure 6 shows how each byte is bit swapped. The Perl script that loads the PROM swaps the bits prior to loading it into the PROM, so that when the data is read back from the PROM, it is read in the correct order.

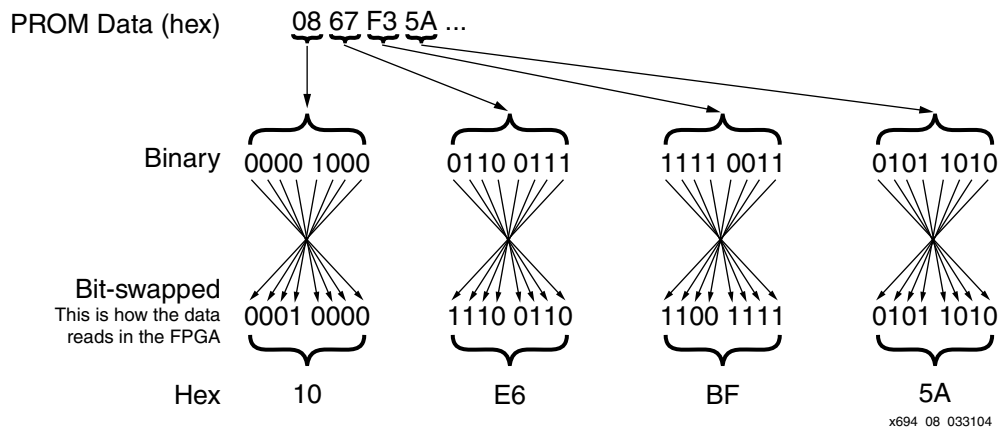


Figure 6: Bit Swapping Output of PROM

Note that the bits are swapped in the MCS file if it is compared to what is being read in from the PROM.

[Table 2](#) is the truth table on how the PROM control inputs get data output from the PROM.

Table 2: PROM Control Input Truth Table

Control Inputs		Internal Address	Outputs
OE/RESET	CE		DATA
High	Low	If address <= TC: increment If address > TC: don't change	Active
Low	Low	Held reset	High-Z
High	High	Held reset	High-Z
Low	High	Held reset	High-Z

Firmware Performance

Performance of the boot operation is slow because the data is read serially and the PROM clock is generated in software. The time to access the PROM is also affected by the size of the bitstream stored in the PROM. If the size of the bitstream stored in the PROM is large, then it takes a long time for the promread function to parse through the PROM until it finds the software or user data sections in the PROM.

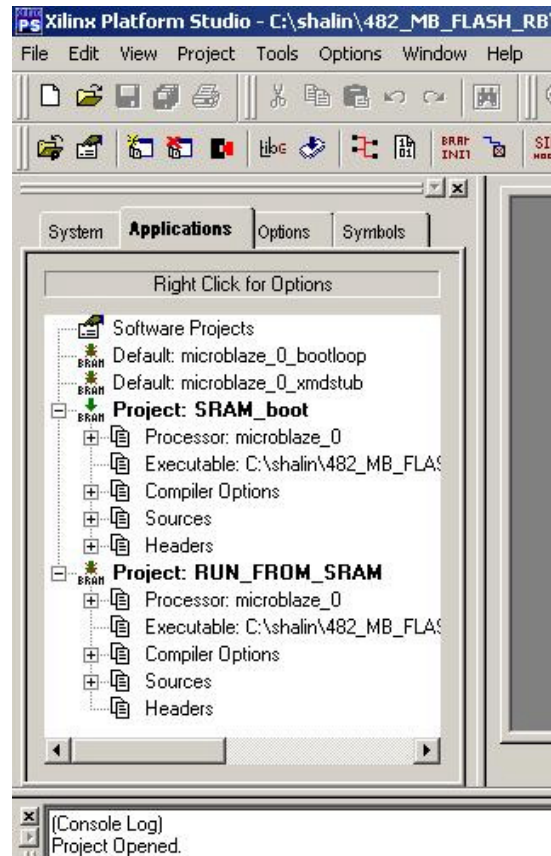
Real-time performance was benchmarked on the Spartan-3 Starter Kit board running at 50 MHz. A 1 Mbit bitstream took approximately two seconds to parse through the PROM.

The size of the software code for the promread function is 0x344 (or 836) bytes.

Dual Software Project

In many embedded systems, designers use linker scripts to divide sections of software code into different memories. Another approach is to use multiple software projects based on the code executed. This reference design uses a two software project concept to divide the boot loader software and the application software. The boot loader software resides in and executes from the block RAM, and the application software resides in and executes from the SRAM. You must set up the SRAM program so that it does not get initialized into the block RAMs. Upon boot-up, the boot loader copies data from the PROM into the SRAM. At the completion of the copy, the boot loader jumps to the start of the SRAM to begin execution of the application's software. The jump to the SRAM is done using a function pointer as described in ["Changing the Program Counter \(PC\) in a C Program."](#)

[Figure 7](#) shows the setup of the dual software project in EDK 6.2i.



X482_07_072804

Figure 7: Dual Software Project Approach

Changing the Program Counter (PC) in a C Program

At the end of a boot loader, it is common to jump from the boot program space to another address location to begin execution of application instructions. In such an example, you can simply change the PC using assembly instructions; however, this change results in mixed languages if your software is written in C. A simple solution is to use a function pointer in C, as outlined in Figure 8. You must set `PROG_START_ADDR` to the start address of the applications software in the boot loader.

```
//declare before main()
// Function point that is used at the end of the program
// to jump to the address location stated by PROG_START_ADDR
#define PROG_START_ADDR 0x80180000
int (*func_ptr) ();

// declare after main()
// function point that is set to point to the address of
// PROG_START_ADDR
func_ptr = PROG_START_ADDR;
// jump to start execution code at the address
// PROG_START_ADDR
func_ptr();
```

Figure 8: C-coded Function Pointer Used to Create a Jump Instruction in Assembly

Figure 9 shows how the C code in Figure 8 is converted to MicroBlaze assembly language.

```

84     func_ptr = PROG_START_ADDR;
- 0xe4 <main+16>:    imm    -32744
- 0xe8 <main+20>:    addik  r3, r0, 0
- 0xec <main+24>:    swi    r3, r0, 1808// 0x710 <func_ptr>
86     func_ptr();
- 0xf0 <main+28>:    brald  r15, r3
- 0xf4 <main+32>:    or     r0, r0, r0

```

Figure 9: Disassembly of Function Pointer

Reducing Code Size by Changing the Boot Code (optional)

Code size reduction, as described in this section, is accomplished by removing the default interrupt and exception handlers inserted by the EDK tools for the boot-loader software project named `SRAM_boot`. This optimization is *optional* and is only needed when software code needs to be reduced in size, and interrupts and exceptions are not needed during the boot-loading process. This section describes how to change the initialization files and integrate them into your project. It also indicates the potential software savings that can be seen when modifying these files. If optimization is used, then the interrupt and exception handlers may still remain in the `RUN_FROM_SRAM` software project, running from SRAM once it is boot loaded. However, additional steps are necessary to set up the processor to access these handlers, as described in [“Resetting the Reset, Exception, and Interrupt Handlers.”](#) For more information on the software initialization file, refer to the [Embedded System Tools Guide](#).

The `boot.s` and `critinit.s` initialization files are included with this reference design. To modify these initialization files, follow these steps:

1. Add the two files to the software project.
2. Add a linker script.
3. Modify the compiler options to disable the automatic insertion of the initialization files.

The `critinit.s` file keeps track of global variables. You must not modify this file unless you have a strong understanding of it and want to disable the use of global variables.

The `boot.s` file has the handler's for interrupts and exceptions to `main()`. Figure 10 shows the two sections that can be commented out. By commenting out the Exception handler section, a design saves 28 bytes, and by commenting out the Interrupt handler, a design saves 284 bytes. These sections might not be needed if a design or software project does not require the use of these functions. Note that by adding the `boot.s` and `critinit.s` files to the software project, the tools automatically compile and link the files using the assembler and the linker if certain options are set in the linker script and compiler options.

```

/* Set Exception Handler */
la    r6,r0,_exception_handler
sw    r6,r1,r0
lhu   r7,r1,r0
shi   r7,r0, 0xa    # [imm for exception]
shi   r6,r0, 0xe    # [lower half for exception ]*/

/* Set Interrupt Handler */
la    r6,r0,_interrupt_handler
sw    r6,r1,r0
lhu   r7,r1,r0
shi   r7,r0, 0x12   # [imm for exception]
shi   r6,r0, 0x16   # [lower half for interrupt ]

```

Figure 10: Interrupt and Exception Handling in the C Runtime File (CRT)

Figure 11 shows the lines in the linker script that need to be added or changed in **bold**. The initialization files are placed in the section called "boot", and there are up to 0x1FF bytes of memory allocated for it. This example assumes a memory space of 16 KB (0x0 to 0x3FFF). This size must be modified to match the memory space in every specific design based on the design's Microprocessor Hardware Specification (MHS) file. In this linker script, the contents of the .boot section are placed in the memory space 0x0 to 0x7FF, and all other software code is placed in the ilmb_cntrl section from 0x200 to 0x3FFF.

```

/* Define all the memory regions in the system */
MEMORY
{
  ilmb_cntrl : ORIGIN = 0x00000200, LENGTH = 0x3fff - 0x01ff
  boot : ORIGIN = 0x00000000, LENGTH = 0x01ff
}
.
.
.
.boot : {*(.boot);} > boot
}

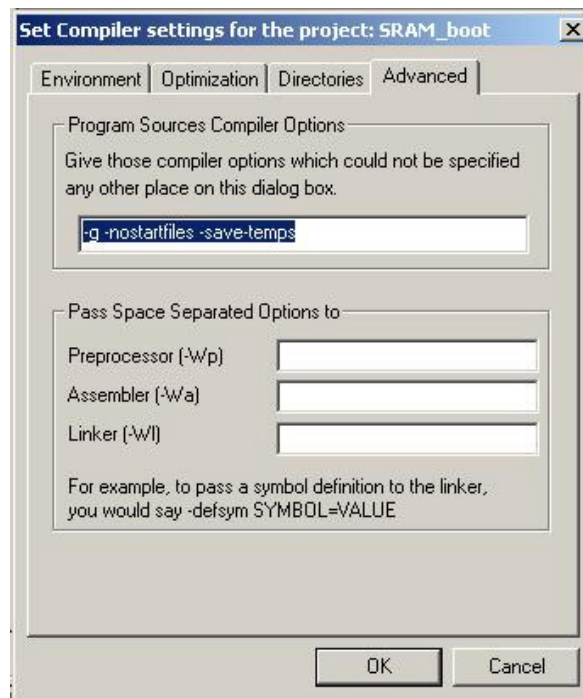
```

Figure 11: Linker Script Changes to be Made for Changes in the CRT

Two compiler settings need to be set to capture the changes to the initialization files:

1. The `-nostartfiles` setting tells the assembler not to include the default initialization files. If this option is forgotten, then a multiple section declaration occurs within the linker when the initialization files are included in the software project.
2. The `-save-temps` setting allows the linker to pick up the handlers from the assembly file.

Figure 12 shows how to set the compiler options.



X482_12_072704

Figure 12: Compiler Changes to Replace the CRT Files

Resetting the Reset, Exception, and Interrupt Handlers

In a dual software project system, once a software project has jumped into SRAM, you may want to access the interrupt and exception handlers that are stored in the SRAM. Usually in a MicroBlaze system, the block RAM is memory mapped to 0x0, and the SRAM resides somewhere else in the memory mapping. However, by default, a MicroBlaze system jumps to either address 0x0 upon any reset or address 0x10 upon an interrupt. The default MicroBlaze handler addresses are as follows:

- Reset: 0x0
- Exception: 0x8
- Interrupt: 0x10

To resolve this issue, you can write assembly jump routines at the default MicroBlaze handlers to jump to the location of the software handlers in SRAM as shown in [Figure 13](#). The software handlers are taken, and then the jump instruction is inserted at 0x0, 0x8, or 0x10 for the Reset, Exception, or Interrupt handling, respectively. Note that this approach dynamically modifies instructions and may not be acceptable for some software designers; however, this is one solution to maintaining separate boot loaders and software code. The benefit to the approach described in this application note is that the boot loader does not need any knowledge about the applications software, however, if the boot loader knows the locations of interrupt and exception handlers in the SRAM, dynamically modifying software is not necessary in the applications software.

```
//insert before main()
extern int _start;
extern int _exception_handler;
extern int __interrupt_handler;

//=====

//insert after main() and after variable declarations
int x = &_start;
*(int*)(0x0) = 0xb0000000 | (((x-1) & 0xFFFF0000) >> 16 );
*(int*)(0x4) = 0xb8000000 | (((x-1) & 0xFFFF));

x = &_exception_handler;
*(int*)(0x8) = 0xb0000000 | (((x-1) & 0xFFFF0000) >> 16 );
*(int*)(0xB) = 0xb8000000 | (((x-1) & 0xFFFF));

x = &__interrupt_handler;
*(int*)(0x10) = 0xb0000000 | (((x-1) & 0xFFFF0000) >> 16 );
*(int*)(0x14) = 0xb8000000 | (((x-1) & 0xFFFF));
```

Figure 13: Dynamic Software to Rewrite the Reset, Exception, and Interrupt Handlers for a MicroBlaze System

Usage/Flow

The creation of the Platform Flash/PROM boot loader requires the use of custom scripts and flows. This section describes the flow and the usage of the scripts to accomplish this. This section also describes how to populate an MCS file with the contents of an executable linking format (ELF) file for the software to be run out of SRAM or with user data. [Table 3](#) shows the features available through the customer scripts provided in the reference design.

Table 3: Features of the Provided Perl Scripts

Format Contents	MEM Address and Data
Size / 16 bytes of data → Binary stored	16 bytes
Address block overhead	8 bytes
Start Address Stored	No
Address Location Stored	Yes
Checksum	No
Boot code	Medium
Multiple ELF file support	Yes
Flow	Three steps: gcc → Data2MEM → pc_ublaze.pl

Creating an MCS File

All flows start with an MCS file. The MCS file can be created using iMPACT or promgen. Refer to the respective documentation on how to create an MCS file.

Adding Software Sections to MCS Files

Figure 14 shows the software flow for adding code to PROM files.

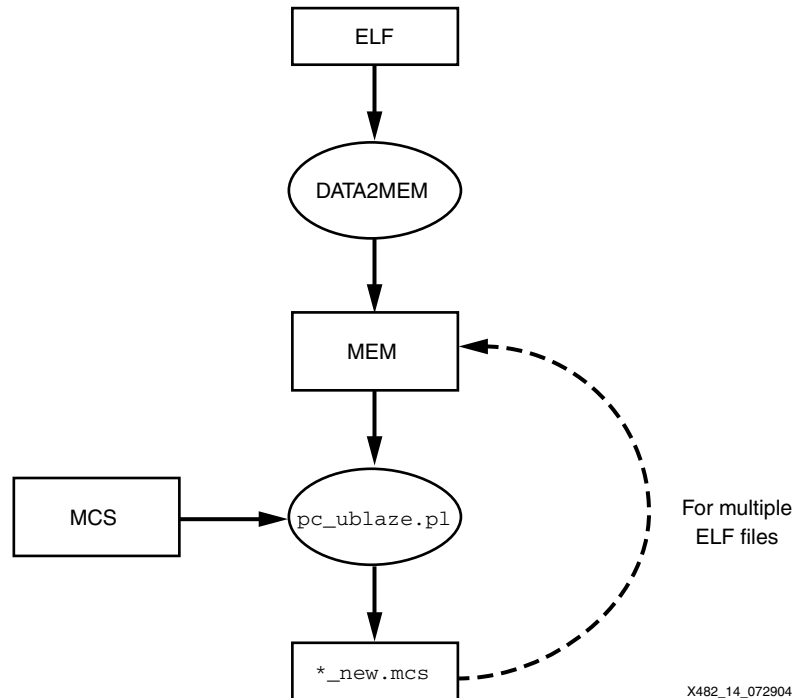


Figure 14: Flow for Adding Application Software to a PROM File

After you have compiled the code to be executed from SRAM, the ELF file is input into Data2MEM to output a MEM file. The encrypted ELF file format is converted to a hex MEM file to be used by the provided Perl script. The command line to create a MEM file from an ELF file is:

```
Data2MEM -bd *.elf -d -o m *.mem
```

For details on running Data2MEM, refer to the [Development System Reference Guide](#).

The next step is to use the provided Perl script `pc_ublaze.pl` to combine the MEM file contents with the MCS file. Note that `xilperl` is a command line tool shipped with the Xilinx ISE software which can run this Perl script.

```
Xilperl pc_ublaze.pl -f mcs -swap on -uf *.mem -pf *.mcs
```

The output of the above command line is `*_new.mcs`, which can be used to program the PROM. The above steps can be repeated to add additional address sections to the MCS file.

Adding a User Data Section to MCS Files

To add a section of user data to the MCS file, you can run the command:

```
Xilperl pc_ublaze.pl -f mcs -swap on -uf userdata.txt -pf *.mcs
```

You must populate the `userdata.txt` file and maintain the following specific requirements:

1. Every data line must be 16 bytes long.
2. Every number must be represented in hex.
3. To add a comment, insert a '#' before the line.
4. Put a synchronization word at the start of the data section. In the example below, the default synchronization word is `0x8F9FAFBF`.

```
#This is data block 0
#The sync pattern is 8F9FAFBF
#The data is ASCII code for:
#XAPP 694 DATA BLOCK 0
#0123456789012345678901234567890
8F9FAFBF584150502036393420444154
4120424C4F434B203000000000000000
```

Perl Script Details

The use models for the Perl script were described above.

Care must be taken to not add too much user-defined data to the PROM, otherwise the configuration tools reject the PROM file. To select a PROM that can store both the FPGA configuration and the user-defined data, simply add the number of bits used for the FPGA configuration to the number of user-defined data bits, bits of software code, and synchronization patterns overhead. The number of bits used for the FPGA configuration can be found by consulting the appropriate FPGA data sheet.

If the address synchronization word of the address section needs to be modified, then the corresponding line must be changed in the Perl script. The data synchronization word is already set by the user in the `userdata.txt` file. Therefore nothing needs to be changed in the Perl script as shown in [Figure 15](#).

```
//Address Synchronization word is set to 0x9F8FAFBF
# Software section synchronization word
$sync_word = "9F8FAFBF";
# user data section synchronization word
$sync_word_2 = "8F9FAFBF";
```

Figure 15: Synchronization Words from the Perl Scripts

The Perl script has a check for the synchronization words (`0x8F9FAFBF` and `0x9F8FAFBF`). It issues a warning if either synchronization word is found. If this is your first time running the script with a new MCS file generated by `iMPACT`, then a new synchronization word must be chosen. If you are appending multiple sections to an MCS file as shown by the dotted lines in

[Figure 14](#) and you did not get an error the first time you ran the script, then the warning can be ignored.

Conclusion

This application note describes board-level changes to be made to read a PROM after FPGA configuration, ways to hold multiple data streams in a PROM, software to read user data from a PROM, boot-loading methodologies for software systems, methods to optimize MicroBlaze hardware and software systems for a boot loader, and finally software flows to enable appending of PROM files with software and user data. These concepts are all applied to help reduce overall system cost of a deployed MicroBlaze system.

Design Resources

The reference design described in this application note can be downloaded from the following link:

<http://www.xilinx.com/bvdocs/appnotes/xapp482.zip>

References

The following Xilinx documents provide supplementary material useful with this application note:

1. [XAPP694](#): “Reading User Data from Configuration PROMs”
2. [XAPP501](#): “Configuration Quick Start Guidelines”
3. [XAPP138](#): “Virtex FPGA Series Configuration and Readback”
4. [UG130](#): *Spartan-3 Starter Kit Board User Guide*
5. [UG111](#): *Embedded System Tools Guide*
6. [Development System Reference Guide](#)
7. [DS099](#): Spartan-3 FPGA Family Complete Data Sheet
8. [MicroBlaze Processor Reference Guide](#)
9. [DS123](#): Platform Flash In-System Programmable Configuration PROMs

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
08/19/04	1.0	Initial Xilinx release.